Modeling self-adaptative IoT architectures

Iván Alfonso[1,2], Kelly Garcés[1], Harold Castro[1], and Jordi Cabot[2,3]

[1] Department of Systems and Computing Engineering, Universidad de los Andes, Bogotá, Colombia
{id.alfonso,kj.garces971,hcastro}@uniandes.edu.co
[2] Universitat Oberta de Catalunya, Barcelona, Spain
[3] ICREA
jordi.cabot@icrea.cat

Abstract. An Internet of Things (IoT) architecture describes a network of (physical) objects exchanging data over the Internet. In complex IoT deployments, as those typical of Industry 4.0 systems, this architecture comprises four different layers: device, edge, fog, and cloud. Moreover, the composition and computation tasks assigned to each layer may dynamically evolve due to environmental changes.
As such, modeling IoT architectures is a complex process that must cover as well the specification of self-adaptation rules to ensure the optimized execution of the IoT system. To facilitate this task, we propose a new IoT Domain-Specific Language (DSL) covering both the static and dynamic aspects of an IoT deployment. The definition of the DSL, its projectional-based editor and a first prototype of a Kubernetes manifest generator are available as open source software.

1 Introduction

The emergence of the Internet of Things (IoT) [14] has dramatically changed how physical objects are conceived in our society and industry. In this new IoT era, every object becomes a complex cyber-physical system (CPS) [19], where both the physical characteristics and the software that manages them are highly intertwined. More specifically, IoT is defined by the ITU as a "global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies" [32].

The ideas behind the IoT have been especially embraced by industry in the so-called Industrial IoT (IIoT) or Industry 4.0 [21]. Currently billions of devices become connected with potential capabilities to sense, communicate, and share information about their environment [10]. Traditional IoT systems rely on cloud-based architectures, which allocate all processing and storage capabilities to cloud servers. Although cloud-based IoT architectures have advantages such as reduced maintenance costs and application development efforts, they also have limitations in bandwidth and communication delays [20]. Given these limitations, edge and fog computing have emerged with the goal of distributing processing and storage close to data sources (i.e. things). Today, developers tend to leverage the advantages of edge, fog, and cloud computing to design multi-layered architectures that meet IoT system requirements.

Nevertheless, creating such complex designs is a challenging tasks. Even more challenging is managing, and adapting IoT systems at runtime to ensure the optimal performance of the system while facing changes in the environment conditions. IoT systems are commonly exposed to changing environments that induce unexpected events at run-time (such as unstable signal strength, latency growth, and software failures) that can impact its Quality of Service (QoS). To deal with such events, a number of runtime strategies (such as auto-scaling and offloading tasks) should be automatically applied.

In this sense, a better support to define complex IoT systems and their (self)adaptation rules to semi-automate the deployment and evolution process is necessary [26]. A usual strategy when it comes to modeling complex domains is to develop a domain-specific language (DSL) for that domain [6]. In short, a DSL offers a set of abstractions and vocabulary closer to the one already employed by domain experts, facilitating the modeling of new systems for that domain and avoiding the syntactic disorder that usually occurs when using a GPL [11].

Nevertheless, current DSLs for IoT do not typically cover multi-layered architectures [23, 24, 33, 9] and even less include a sublanguage to ease the definition of the dynamic rules governing the IoT system.

Our work aims to cover this gap by proposing a new domain-specific language (DSL) for IoT systems focusing on three main contributions: (1) modeling primitives covering the four layers of an IoT system, including IoT devices (sensors or actuators), edge, fog, and cloud nodes; (2) modeling the deployment and grouping of container-based applications on those nodes; and (3) a specific sublanguage to express adaptation rules to guarantee QoS at runtime. We have implemented the DSL using the Meta Programming System (MPS)[4]. A proof of concept of a generator for deploying the modeled IoT system on a Kubernetes[5]-based infrastructure is also provided.

The remainder of the paper is organized as follows: Section 2 presents background information. Section 3 presents the DSL design. Section 4 summarizes the related work to this research. In Section 5, we present the DSL implementation and code generation. Finally, Section 6 summarizes the conclusions and future work.

2 Background and Running Example

Multi-layered architectures have emerged to increase the flexibility of pure cloud deployments and help meeting non-functional IoT system requirements. In particular, edge computing and fog computing are solutions that propose to bring computation, storage, communication, control, and decision making closer to IoT devices [16]. There are strong similarities between these two solutions, but

[4] https://www.jetbrains.com/mps/
[5] https://kubernetes.io/

the main difference is where the processing takes place. While edge computing takes place on devices directly connected to sensors and actuators, or on gateways physically close to these, fog computing takes place on LAN-connected nodes usually farther away from the sensors and actuators. Sensors and actuators compose the device layer. Sensors generate information by monitoring physical variables (such as temperature, motion, oxygen, and humidity) and actuators are devices capable of transforming energy into activation of a process to modify a procedure. Valves, motors, and fans are some of the common actuators.

We will use a Smart Building scenario as a running example to better illustrate these concepts. Smart buildings seek to optimize different aspects such as ventilation, heating, lighting, energy efficiency, etc [22]. Adopting the concept of smart building, a hotel company wants to reduce fire risks by automating disaster management in its hotels. A fire alarm and monitoring system are implemented in each of the company's hotels. For the sake of the example, we will assume that all buildings (hotels) have three floors with two rooms each. Fig. 1 presents an overview of the 1st floor of this building. According to this, the infrastructure and software features of the company hotel IoT system are:

Device layer. Each room has a temperature sensor, two smoke sensors, and a fire water valve. Furthermore, an alarm is deployed on each floor.

Edge layer. In each room, an edge node receives the information collected by the sensors of the device layer. This node runs the containerized application (App1) for analyzing sensor data in real time to check for the presence of smoke and generate an alarm state that activates the actuators.

Fog layer. The building has a fog node that communicates with the Edge nodes. This fog node located int the 1st floor of the building, runs a machine learning model to predict fires on any of the building's floors (App2).

Cloud layer. The cloud layer has a server or cloud node that runs two containers: (DB1) a database to store part of the sensor data, and (App3) a web application to display historical information of sensor data and of fire incidents in any of the hotels property of the company.

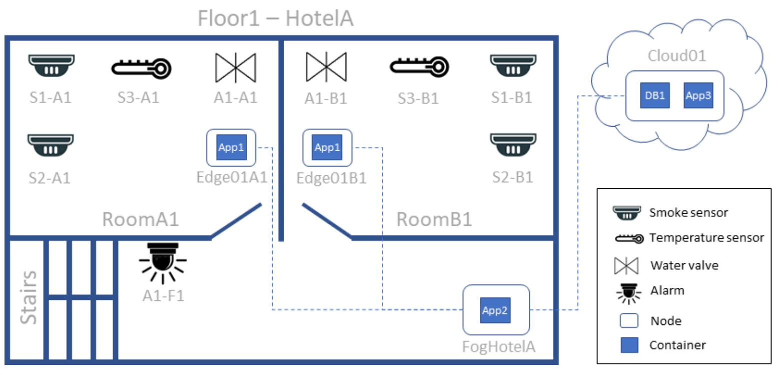


Fig. 1. Overview of the smart building IoT system, first floor

During system operation, QoS must be guaranteed mainly for critical applications. For example, (App1) real-time analysis to detect and alarm a fire should always be available. However, some environmental factors could generate unexpected events that affect system operation. In these cases, the system must self-adapt to guarantee its operation. For instance, if the edge node fails, it is necessary to migrate container App1 to another suitable node. Next section shows how we are able to model all these concepts.

3 DSL for the specification of IoT systems

A DSL is defined by three core ingredients [6]: the abstract syntax (i.e., the concepts of the DSL and their relationships), the concrete syntax (the notation to represent those concepts) and its semantics which most of the time is not formalized.

In this section, we present the DSL for modeling the static and dynamic aspects of the IoT system as follows. First, Section 3.1 describes the abstract syntax and the concrete syntax of the DSL elements for the specification of static aspects including architecture and deployment of containerized applications. Then, Section 3.2 covers the dynamic ones, i.e. the self-adaptation rules.

3.1 Modeling of the IoT architecture

Abstract syntax The abstract syntax of the DSL is commonly defined through a metamodel that represents the domain concepts and their relationships. Fig. 2 shows the excerpt of the metamodel that abstracts the concepts to define multi-layer IoT architectures. The sensors and actuators of the device layer are modeled using the *Sensor* and *Actuator* concepts that inherit from the *IoTDevice* concept. This generalization is restricted to be complete and disjoint, each instance of *IoTDevice* must be either a *Sensor* or an *Actuator* but not both. All *IoTDevice*s have a connectivity type (such as ethernet, wifi, ZigBee, or another) represented as an attribute with the *Connectivity* enumeration as type. Because the MQTT messaging protocol is becoming the standard for M2M communications [1], we cover the concepts for specifying this type of communication between IoT devices and nodes. Therefore, *IoTDevices* are publishers or subscribers to a topic MQTT broker specified by the attribute *topic*). The gateway of an *IoTDevice* can be modeled through the *gateway* relationship with *EdgeNode* concept. Via this gateway, the sensor can communicate with other nodes, e.g. the MQTT broker node.

The location of IoTDevices can be specified by means of geographic coordinates (*latitude* and *longitude* attributes). Both *Sensors* and *Actuators* have a type represented by the concepts *SensorType* and *ActuatorType*. For instance, following the example scenario (Fig. 1), there are temperature and smoke type sensors, and there are valve and alarm type actuators.

Physical (or even virtual) spaces such as rooms, buildings, or corridors can be represented by the concept *Region*. A *Region* can contain more regions (relationship *subregions* in the metamodel). For example, region *Floor01* (Fig. 1) contains subregions *RoomA* and *RoomB*. *IoTDevice*s, *EdgeNode*s, and *FogNode*s can be part of regions or subregions. Therefore, the *Edge01A1* node is located in the *RoomA* region of *Floor1* of *HotelA*, while the *FogHotelA* node is located in the *Floor1* region of *HotelA*.

Edge, fog and cloud nodes are all instances of *Node*, one of the key concepts of the metamodel. A node is the concept responsible for hosting the software containers. Every node belongs to a single subtype. Communication between nodes can be specified by means of the *linkedNodes* relationship as sometimes we may want to indicate what nodes on a certain layer could act as reference nodes in another layer (e.g. what cloud node should be the first option for a fog node). Nodes can be also grouped in clusters of nodes (typically of the same type but this is not mandatory) that work together. A *Cluster* has a master node (represented by the *master* relationship) and several worker nodes (represented by the *workers* relationship).

A *Node* can host several software containers according to its capabilities and resources. These resources are represented by the attributes *cpuCores*, *memory*, and *storage*. The *containers* composition relationship represents the containers that are hosted and executed on a *Node*. Each software container runs an application (represented by the concept *Application*) that has a minimum required resources specified by the attributes *minRam* and *minStorage*. The repository of the application image is specified through the *imageRepo* attribute.

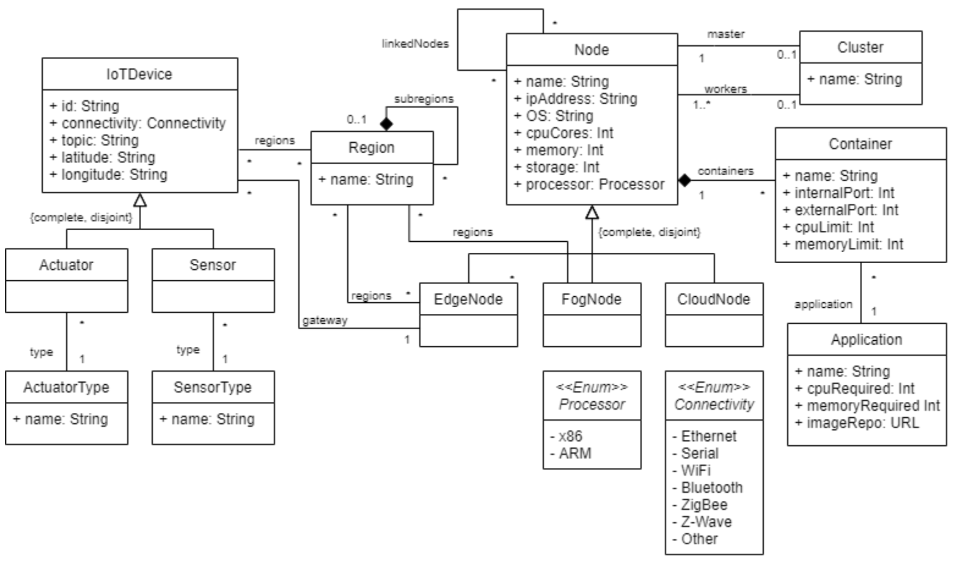
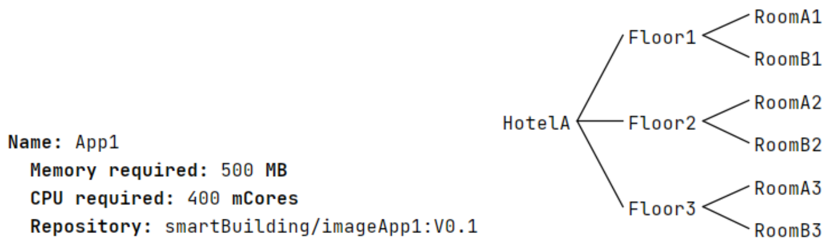


Fig. 2. Excerpt of the metamodel depicting multi-layer architecture and deployment

Concrete syntax. The concrete syntax refers to the type of notation (such as textual, graphical, tabular, or hybrid) to represent the concepts of the metamodel. We take advantage of MPS projectional editors to define a hybrid notation (textual, tabular, and tree view). Projectional editors are editors in which the user's editing actions directly change the Abstract Syntax Tree (AST) without using a parser [3]. That is, while the editing experience simulates that of classical parsing-based editors, there is a single representation of the model stored as an AST and rendered in a variety of perspectives thanks to the corresponding projectional editors that can deal with mixed-language code and support various notations such as tables, mathematical formulas, or diagrams.

Our DSL enables the modeling of all concepts using a textual notation. Additionally, for some concepts we also offer complementary notations that we believe are better suited for that particular concept. In particular, we define a tabular notation for modeling nodes and IoT devices, and a tree notation for regions. An example of the definition one of these projectional editors are presented in Section 5.

Example scenario An excerpt of the Smart building scenario (Section 2) using this part of the DSL is presented next. Fig. 3(a) show the textual modeling of application App1 while Fig. 3(b) describes the regions of the example. *HotelA* represents a region with three subregions (three floors) and two rooms as subregions of each floor.



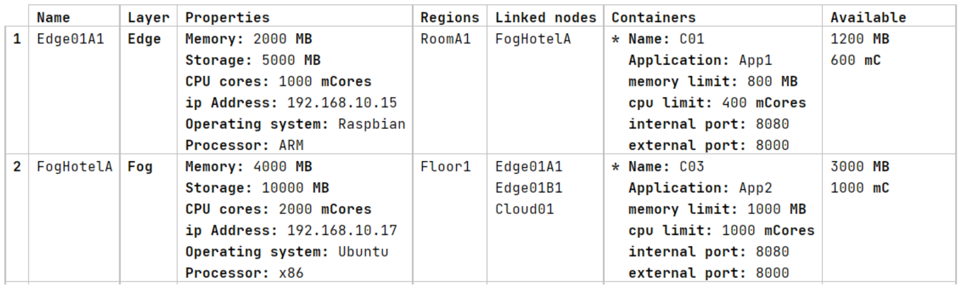
```
Name: App1
  Memory required: 500 MB
  CPU required: 400 mCores
  Repository: smartBuilding/imageApp1:V0.1
```

(a) Application modeling example (b) Regions modeling example

Fig. 3. Modeling example

For nodes, we propose a tabular notation where each row corresponds to a node. Fig. 4 shows the modeling for nodes *Edge01A1* and *FogHotelA*. Each node belongs to a layer (edge, fog, or cloud), has a list of hardware properties (*CPU cores* are specified in milicores, e.g., 1000 mCores = 1 core.), is located in one or more regions, and hosts containers of some previously defined application.

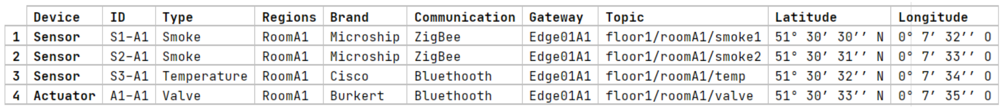
The concept *Cluster* represents a group of nodes modeled by means of a textual notation in a similar way to the applications. To relate the master node and the worker nodes it is necessary to have previously defined them. Finally,



	Name	Layer	Properties	Regions	Linked nodes	Containers	Available
1	Edge01A1	Edge	Memory: 2000 MB Storage: 5000 MB CPU cores: 1000 mCores ip Address: 192.168.10.15 Operating system: Raspbian Processor: ARM	RoomA1	FogHotelA	* Name: C01 Application: App1 memory limit: 800 MB cpu limit: 400 mCores internal port: 8080 external port: 8000	1200 MB 600 mC
2	FogHotelA	Fog	Memory: 4000 MB Storage: 10000 MB CPU cores: 2000 mCores ip Address: 192.168.10.17 Operating system: Ubuntu Processor: x86	Floor1	Edge01A1 Edge01B1 Cloud01	* Name: C03 Application: App2 memory limit: 1000 MB cpu limit: 1000 mCores internal port: 8080 external port: 8000	3000 MB 1000 mC

Fig. 4. Nodes modeling example (tabular notation)

IoT devices can be modeled using a tabular notation. Fig. 5 shows the list of sensors and actuators located in the *RoomA1* region.



	Device	ID	Type	Regions	Brand	Communication	Gateway	Topic	Latitude	Longitude
1	Sensor	S1-A1	Smoke	RoomA1	Microship	ZigBee	Edge01A1	floor1/roomA1/smoke1	51° 30′ 30′′ N	0° 7′ 32′′ O
2	Sensor	S2-A1	Smoke	RoomA1	Microship	ZigBee	Edge01A1	floor1/roomA1/smoke2	51° 30′ 31′′ N	0° 7′ 33′′ O
3	Sensor	S3-A1	Temperature	RoomA1	Cisco	Bluethooth	Edge01A1	floor1/roomA1/temp	51° 30′ 32′′ N	0° 7′ 34′′ O
4	Actuator	A1-A1	Valve	RoomA1	Burkert	Bluethooth	Edge01A1	floor1/roomA1/valve	51° 30′ 33′′ N	0° 7′ 35′′ O

Fig. 5. IoT devices modeling example (tabular notation)

3.2 Modeling of the Self-Adaptation Rules

The dynamic environment of an IoT system generates unexpected changes that could affect the QoS. The system should be prepared to cope with unexpected changes and self-adapt to different situations. Because of this, our DSL enables the modeling of self-adaptation rules of the system.

Abstract syntax The metamodel excerpt representing the abstract syntax for defining the rules is presented in Fig. 6. Note that we reuse an existing *Expression* language to avoid redefining in our language all the primitive data types and all the operations to manipulate them. Such Expression language could have been for instance the Object Constraint Language (OCL) but to facilitate the implementation of the DSL later on, we directly reused the MPS Baselanguage. As such, the metamodel add two new types of Expressions to facilitate the definition of conditions involving sensor events or QoS metrics. As subexpressions, sensor and QoS conditions can be combined also with all other types of expressions (e.g. numerical ones) in a complex conditional expression.

Every rule is an instance of *AdaptationRule* that has a triggering condition expressed as discussed above. More specifically, a *QoSEvent* condition is a relational expression that represents a threshold of resource consumption metrics or QoS while a *SensorEvent* represents the occurrence of an event resulting from the analysis of sensor data (for example, the detection of fire in a room). Note that SensorEvent conditions can be linked to specific sensors or to sensor types to express conditions involving a group of sensors.

Similarly, *QoSEvent* type condition allows to check a Metric (such as Latency, CPU consumption, and others) on a specific node or a group of nodes belonging to a *Region* or *Cluster*. For example, the condition $CPU(Floor1.edgeNodes) > 50$ means that it is triggered when the CPU consumption on the edge nodes of *Floor1* exceeds 50%. The *SensorEvent* type condition allows to check the data of a specific sensor or a group of sensors located in a *Region*. For example, the condition $Room1.Temperature(2) > 40$ means that it is triggered when the temperature of two sensors inside *Room1* exceeds 40.

Moreover, we can define that the condition should be true over a certain period (to avoid firing the rule in reaction to minor disturbances) before executing the rule. Once fired, all or some of the actions are executed in order, depending on the *allActions* attribute. If set to false, only the number of *Action*s specified by the attribute *actionsQuantity* must be executed starting with the first one in order and continuing until the required number of actions have been successfully applied.

An action can be classified as *Offloading*, *Redeploy*, or *Scaling*. The *Offloading* action consists in migrating a container from a source node to a destination node. This migration can be between nodes of different layers. The *container* relationship represents the container that will be offloaded. The target node is specified by the *targetNode* relationship. However, if the target node does not have the resources to host the container, a cluster or a group of nodes in a *Region* can be specified (*targetRegion* and *targetCluster* relationships) to offload the container. The *Scaling* action involves deploying replicas of an application. This application to be scaled is represented by the *app* relationship, and the number of replicas to be deployed is defined by the *instances* attribute. The replicas of the application are deployed in one or several nodes of the system represented by the *targetNodes*, *targetCluster*, and *targetRegion* relationships. Finally, the *Redeployment* action consists in stopping and redeploying a container running on a node. The container to redeploy is indicated by the *container* relationship.

Concrete syntax The system adaptation rules are specified using a textual notation using as keywords the names of the metaclasses in the abstract syntax. The conditions follow the grammar of a relational expression with the use of mathematical symbols (such as $<$, $>$, and $=$) and logical operators (such as & and ||). It is worth to note that, given that the textual notation is a projection of the abstract syntax, the MPS editor (see Section 5) offers a powerful autocomplete feature that will guide the designer through the rule creation process.

Example Scenario As an example, we show an adaptation rule (Fig. 7) for our hotel scenario to guarantee the execution of the *App1* deployed on the *Edge01A1* node (container *C01*). The modeled adaptation rule offloads the container *C01* hosted on node *Edge01A1* to a nearby node (e.g., node *Edge01B1*) when the

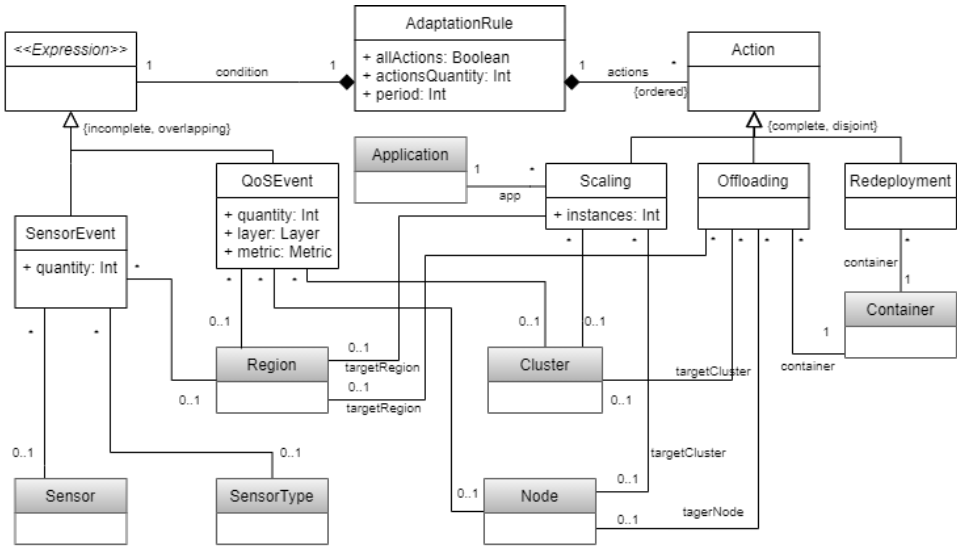
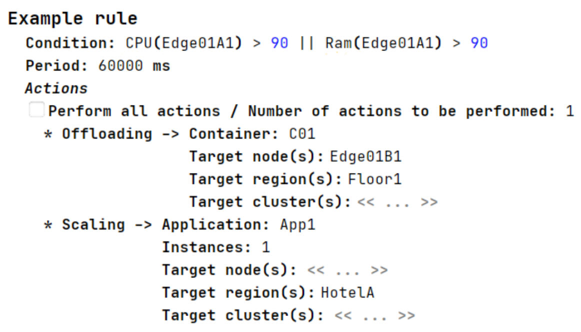


Fig. 6. Excerpt of the metamodel depicting adaptation rules

CPU or RAM consumption exceeds 90% for one minute. If the *Edge01B1* node does not have the necessary resources to host that new container, we can specify a Region (e.g. *Floor1*) so that a suitable node will be searched there. However, if this offloading action cannot be executed, for example, because in *Floor1* there is no node capable of hosting the container, then we must define a backup action. Therefore, we have modeled a second action (*Scaling*) to deploy a new container instance of the *App1* application on any of the nodes of the *HotelA*. For this adaptation rule only one action (the first or the second one) will be executed. Therefore, the checkbox *Perform all actions* must be unchecked and the number of actions to be performed must be set to one.



```
Example rule
  Condition: CPU(Edge01A1) > 90 || Ram(Edge01A1) > 90
  Period: 60000 ms
  Actions
   ☐Perform all actions / Number of actions to be performed: 1
    * Offloading -> Container: C01
                    Target node(s): Edge01B1
                    Target region(s): Floor1
                    Target cluster(s): << ... >>
    * Scaling -> Application: App1
                 Instances: 1
                 Target node(s): << ... >>
                 Target region(s): HotelA
                 Target cluster(s): << ... >>
```

Fig. 7. Adaptation rule modeling example

4 Related Work

Modeling of cloud architectures has been widely studied [27, 28, 31, 4, 5, 2, 31], including provisioning of resources for cloud applications. Nevertheless, these proposals do not cover multi-layered architectures involving fog and edge nodes. This is also the case for Infrastructure-as-code (IaC) tools such as Terraform[6], Pulimi[7], Cloudify[8], or Puppet[9] and other IoT-focused proposals [24].

A few works such as [33, 29, 30, 18] propose solutions for deploying containerized applications on edge and fog nodes using orchestrators such as Kubernetes and Docker Swarm. However, the deployment process is manual and fixed, with no adaptation policies.

Regarding adaptation policies for IoT or cloud architectures, works such as [8, 15, 7, 17, 13] propose partial solutions as they either restrict the parts of the system that could be adapted (e.g. only the global monitoring component to adapt to manual changes in the deployed components) and/or offer some type of adaptation rules but with limited expressiveness and rules that must be manually triggered instead of being self-adaptive.

More similars to our proposal, Lee et al. [23] present a self-adaptive framework to dynamically satisfy functional requirements for IoT systems. This framework, based on MAPE loop, enables modeling of the IoT environment through a finite-state machine. However, adaptations are only enabled at the device layer level (e.g., opening a window when the temperature exceeds a limit). Rules for other layers or involving several layers are not supported. Petrovic et al. [25] propose SMADA-Fog, a semantic model-driven approach to deployment and adaptivity of container-based applications in Fog computing scenarios. However, SMADA-Fog does not allow the specification of complex adaptation rules composed of various conditions and actions. Moreover, grouping nodes and IoT devices according to their location is not possible, forbidding the possibility to apply adaptations on group of nodes belonging to a cluster or a given region.

To sum up, ours is the first proposal that enables the modeling of multi-layer IoT architectures (device, edge, fog, and cloud) and the definition of complex rules covering all layers (and combinations of) and involving multiple conditions and actions that can, potentially, involve groups of nodes in the same region or cluster of the IoT system.

5 Tool Support

Our DSL is implemented using MPS, an open-source language workbench developed by Jetbrains. We briefly describe the creation of the modeling environment for our DSL and that of the proof-of-concept code generator. Both are freely available in our repository.

[6] https://www.terraform.io/
[7] https://www.pulumi.com/
[8] https://cloudify.co/
[9] https://puppet.com/

5.1 Modeling editor

To develop the modeling environment for the DSL, we have to define in MPS three core elements: structure, editors and constraints

Structure Structure defines the abstract syntax of the DSL by defining all metamodel concepts such as *Node*, *AdaptationRule*, and *Region*. In MPS, each concept has properties (attributes), children (composition relationships), and references. Concepts can extend from other concepts to represent inheritance relationships. For example, the *EdgeNode* concept extends from the *Node* concept.

Editors Projection editors define the AST code editing rules. They enable the definition of textual, graphical, tabular, and other types of notation for a concept. Not all concepts require all projections, this is a language design decision. Similarly, each type of projection can be configured for every specific concept. We have defined textual, tabular, and tree view editors by implementing the mbeddr[10] extension of MPS. For example, Fig. 8 shows the tabular editor for modeling the *Sensor* concept. We use the *partial table* command to define the table structure (cells, content, and column headers). By defining this editor, the user is enabled to model *Sensors* using a tabular notation as shown in Fig. 5.

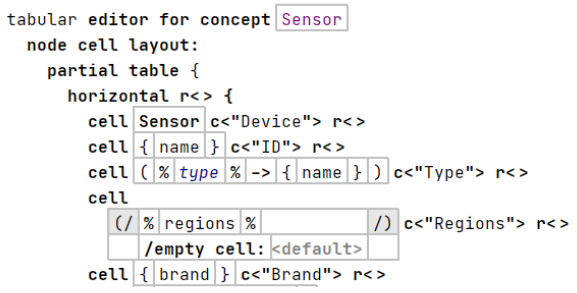
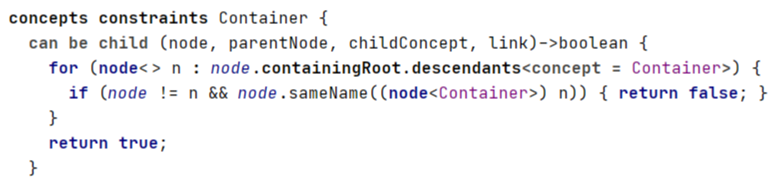

```
tabular editor for concept Sensor
  node cell layout:
    partial table {
      horizontal r<> {
        cell Sensor c<"Device"> r<>
        cell { name } c<"ID"> r<>
        cell ( % type % -> { name } ) c<"Type"> r<>
        cell
          (/ % regions %        /) c<"Regions"> r<>
             /empty cell: <default>
        cell { brand } c<"Brand"> r<>
```

Fig. 8. Definition of the tabular editor for the *Sensor* concept

Constraints Constraints restrict the relationships between nodes as well as the allowed values for properties. In particular, we have used this constraint mechanism to embed in the editor several well-formedness rules required in our DSL specification. For instance, we have added a constraint to avoid repeated names, a constraint to limit the potential values of some numerical attributes, a constraint to restrict the potential relationships between nodes, etc. As an example, the constraint to avoid repeated names for containers is shown in Fig. 9. We use

[10] http://mbeddr.com/

the *can be child* method to check whether instances of the *Container* concept can be hooked as child nodes of other nodes. MPS invokes this method whenever a Container node is modified in the AST, and returns false (i.e., Container not allowed) if its name is repeated with that of another Container.



```
concepts constraints Container {
  can be child (node, parentNode, childConcept, link)->boolean {
    for (node<> n : node.containingRoot.descendants<concept = Container>) {
      if (node != n && node.sameName((node<Container>) n)) { return false; }
    }
    return true;
  }
```

Fig. 9. Container concept to check for repeated names

5.2 Code Generator

While the DSL formalizes the knowledge about the business in the domain, the code generator encapsulates the implementation of that knowledge in a particular target technology. More specifically, as a proof of concept, we have implemented a model-to-text transformation that generates a YAML handle code to deploy container-based applications with the kubernetes orchestrator from our IoT system definition.

To accomplish this, we have used the MPS TextGen component to define and execute the model-to-text transformation. As part of the transformation, a source file per node storing its generated textual representation is saved on disk. The mapping adds to the source file the description of an object deployment with the description of a pod (computation unit of kubernetes representing the collection of containers and storage) and an object service to make the application available via a NodePort. These two objects (deployment and service) can be declared in a YAML file. Due to space limitations, we do not show here the transformation nor the generated code. You can find all this information in the project repository[11], which contains the paper with the appendices and the source code of the project in MPS.

6 Conclusion and Future Work

We have presented a DSL for modeling multi-layered architectures of IoT systems and their self-adaptation rules. The DSL is implemented as a projectional editor created with the Jetbrains MPS tool. This gives us the flexibility to offer, and mix, a variety of concrete notations for the different concepts of the DSL. We have also presented a prototype Kubernetes manifest generator for deploying container-based applications.

[11] https://github.com/SOM-Research/selfadaptive-IoT-DSL.git

As further wok, we will extend the DSL with primitives to express more complex adaptation strategies and patterns (e.g., canary deployment and rolling update [12]). At the tool level, we will work on a visual renderer of the modeled architecture to complement the current projections and complete the code generator to cover the dynamic aspects of the IoT design, potentially by extending a policy engine with adhoc customizations to cover our multi-layered representation. Finally, we plan to validate the DSL in the mining industry (sector providing the funding for this research as well) where IoT monitoring and control systems are heavily used to improve work safety.

References

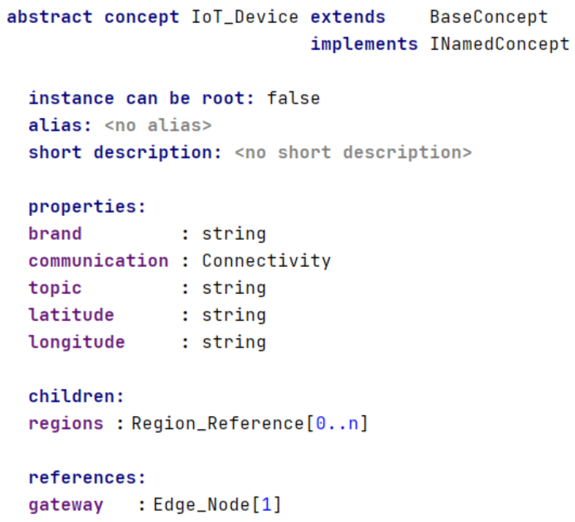
1. Amoretti, M., Pecori, R., Protskaya, Y., Veltri, L., Zanichelli, F.: A scalable and secure publish/subscribe-based framework for industrial iot. IEEE Transactions on Industrial Informatics (2020)
2. Artac, M., Borovšak, T., Di Nitto, E., Guerriero, M., Perez-Palacin, D., Tamburri, D.A.: Infrastructure-as-code for data-intensive architectures: A model-driven development approach. In: IEEE International Conference on Software Architecture (ICSA). pp. 156–15609. IEEE (2018)
3. Berger, T., Völter, M., Jensen, H.P., Dangprasert, T., Siegmund, J.: Efficiency of projectional editing: A controlled experiment. In: Proc. of the 24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering. pp. 763–774 (2016)
4. Bergmayr, A., Breitenbücher, U., Kopp, O., Wimmer, M., Kappel, G., Leymann, F.: From architecture modeling to application provisioning for the cloud by combining uml and tosca. In: CLOSER (2). pp. 97–108 (2016)
5. Bergmayr, A., Rossini, A., Ferry, N., Horn, G., Orue-Echevarria, L., Solberg, A., Wimmer, M.: The evolution of cloudml and its manifestations. In: Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE). pp. 1–6 (2015)
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis lectures on software engineering **3**(1), 1–207 (2017)
7. Chen, W., Liang, C., Wan, Y., Gao, C., Wu, G., Wei, J., Huang, T.: More: A model-driven operation service for cloud-based it systems. In: IEEE International Conference on Services Computing (SCC). pp. 633–640. IEEE (2016)
8. Cianciaruso, L., di Forenza, F., Di Nitto, E., Miglierina, M., Ferry, N., Solberg, A.: Using models at runtime to support adaptable monitoring of multi-clouds applications. In: 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 401–408. IEEE (2014)
9. Ciccozzi, F., Spalazzese, R.: Mde4iot: supporting the internet of things with model-driven engineering. In: International Symposium on Intelligent and Distributed Computing. pp. 67–76. Springer (2016)
10. Cisco, U.: Cisco annual internet report (2018–2023) white paper (2020)
11. Czarnecki, K.: Overview of generative software development. In: Int. Workshop on Unconventional Programming Paradigms. pp. 326–341. Springer (2004)
12. DevOps, X.E.: Best practices for devops: Advanced deployment patterns. url: https://www.wsta.org/wp-content/uploads/2018/09/Best-Practices-for-DevOps-Advanced-Deployment-Patterns.pdf (2018)
13. Erbel, J., Korte, F., Grabowski, J.: Comparison and runtime adaptation of cloud application topologies based on occi. In: CLOSER. pp. 517–525 (2018)

14. Evans, D.: The internet of things: How the next evolution of the internet is changing everything. CISCO white paper **1**(2011), 1–11 (2011)
15. Ferry, N., Chauvel, F., Song, H., Rossini, A., Lushpenko, M., Solberg, A.: Cloudmf: Model-driven management of multi-cloud applications. ACM Transactions on Internet Technology (TOIT) **18**(2), 1–24 (2018)
16. Group, O.C.A.W., et al.: Openfog reference architecture for fog computing. OPFRA001 **20817**, 162 (2017)
17. Holmes, T.: Facilitating migration of cloud infrastructure services: A model-based approach. In: CloudMDE@ MoDELS. pp. 7–12 (2015)
18. Islam, J., Harjula, E., Kumar, T., Karhula, P., Ylianttila, M.: Docker enabled virtualized nanoservices for local iot edge networks. In: IEEE Conference on Standards for Communications and Networking (CSCN). pp. 1–7. IEEE (2019)
19. Jazdi, N.: Cyber physical systems in the context of industry 4.0. In: IEEE Int. Conference on Automation, Quality and Testing, Robotics. pp. 1–4. IEEE (2014)
20. Jiang, Y., Huang, Z., Tsang, D.H.: Challenges and solutions in fog computing orchestration. IEEE Network **32**(3), 122–129 (2017)
21. Kagermann, H., Helbig, J., Hellinger, A., Wahlster, W.: Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group. Forschungsunion (2013)
22. Latifah, A., Supangkat, S.H., Ramelan, A.: Smart building: A literature review. In: Int. Conference on ICT for Smart Society (ICISS). pp. 1–6. IEEE (2020)
23. Lee, E., Seo, Y.D., Kim, Y.G.: Self-adaptive framework based on mape loop for internet of things. sensors **19**(13), 2996 (2019)
24. Patel, P., Cassou, D.: Enabling high-level application development for the internet of things. Journal of Systems and Software **103**, 62–84 (2015)
25. Petrovic, N., Tosic, M.: Smada-fog: Semantic model driven approach to deployment and adaptivity in fog computing. Simulation Modelling Practice and Theory **101**, 102033 (2020)
26. Rhayem, A., Mhiri, M.B.A., Gargouri, F.: Semantic web technologies for the internet of things: Systematic literature review. Internet of Things p. 100206 (2020)
27. Sandobalin, J., Insfran, E., Abrahao, S.: An infrastructure modelling tool for cloud provisioning. In: IEEE International Conference on Services Computing (SCC). pp. 354–361. IEEE (2017)
28. Sandobalin, J., Insfran, E., Abrahão, S.: Argon: A model-driven infrastructure provisioning tool. In: ACM/IEEE 22nd Int. Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 738–742. IEEE (2019)
29. Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Resource provisioning in fog computing: From theory to practice. Sensors **19**(10), 2238 (2019)
30. Scolati, R., Fronza, I., El Ioini, N., Samir, A., Pahl, C.: A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In: Closer. pp. 68–80 (2019)
31. Sledziewski, K., Bordbar, B., Anane, R.: A dsl-based approach to software development and deployment on cloud. In: 24th IEEE International Conference on Advanced Information Networking and Applications. pp. 414–421. IEEE (2010)
32. Union, I.T.: Internet of things global standards initiative (2012)
33. Yigitoglu, E., Mohamed, M., Liu, L., Ludwig, H.: Foggy: a framework for continuous automated iot application deployment in fog computing. In: IEEE International Conference on AI & Mobile Services (AIMS). pp. 38–45. IEEE (2017)

A Definition of concepts in MPS

Each metaclass in our DSL is defined as a *concept* in in MPS. For each concept it is necessary to define several items such as name, properties, and relations. For example, the definition of the *IoTDevice* concept is presented in Fig. 10. This abstract concept has a set of attributes defined as properties, an association relation (*gateway*) with multiplicity of 1, and a smart reference (*regions*). Smart references are the way to define associations with 0..n cardinalities in MPS.

The definition of all the metamodel elements as MPS concepts is available in our public repository.



```
abstract concept IoT_Device extends    BaseConcept
                            implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  brand         : string
  communication : Connectivity
  topic         : string
  latitude      : string
  longitude     : string

  children:
  regions : Region_Reference[0..n]

  references:
  gateway   : Edge_Node[1]
```

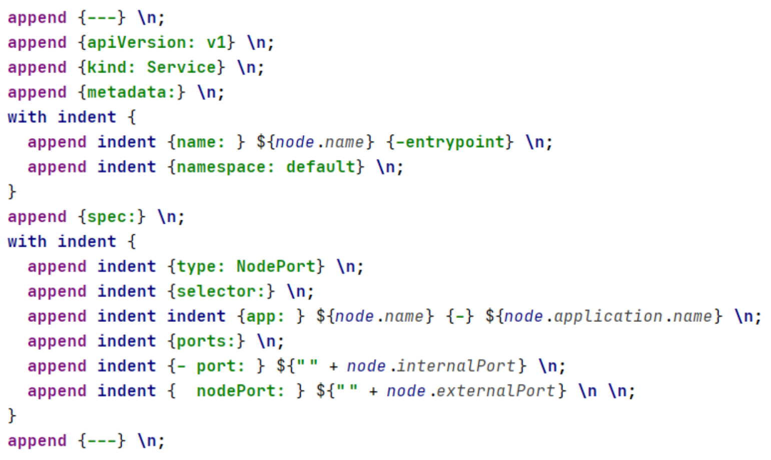
Fig. 10. *IoTDevice* concept definition in MPS

B TextGen and code generation in MPS

The MPS TextGen module supports the definition of model-to-text transformations. We have used it to implement our DSL-to-Kubernetes generator.

More specifically, in our mapping strategy, each modeled container is deployed into a pod. For this, we generate the code of a deployment artifact and a service artifact in a yaml file. An excerpt of the TextGen code for the container concept is as shown in Fig. 11.

Listing 1.1 shows the code generated for the Kubernetes deployment and service of the software container ”C01”. A pod is configured with a container with the repository image *smartBuilding/imageApp1:V0.1* specified by the label *image*; by the label *nodeSelector* the node with the tag *Edge01A1* is assigned to



```
append {---} \n;
append {apiVersion: v1} \n;
append {kind: Service} \n;
append {metadata:} \n;
with indent {
  append indent {name: } ${node.name} {-entrypoint} \n;
  append indent {namespace: default} \n;
}
append {spec:} \n;
with indent {
  append indent {type: NodePort} \n;
  append indent {selector:} \n;
  append indent indent {app: } ${node.name} {-} ${node.application.name} \n;
  append indent {ports:} \n;
  append indent {- port: } ${"" + node.internalPort} \n;
  append indent {  nodePort: } ${"" + node.externalPort} \n \n;
}
append {---} \n;
```

Fig. 11. Excerpt of the TextGen Container concept

host the pod; finally, the application container is exposed via port 8000 (*externalPort* attribute of the *Container* concept).

Listing 1.1. yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: C01–deployment
spec:
  replicas: 1
  selector:
    matchlabels:
      app: C01–App1
  template:
    metadata:
      labels:
        app: C01–App1
    spec:
      containers:
      - name: C01
        image: smartBuilding/imageApp1:V0.1
        ports:
          - containerPort: 8080
    nodeSelector:
      node: Edge01A1
---
apiVersion: v1
kind: Service
metadata:
  name: C01–entrypoint
  namespace: default
spec:
  type: NodePort
  selector:
    app: C01-App1
  ports:
  - port: 8080
    nodePort: 8000
```